Type Expressiveness, Spec and the Separation of Behavior Programming and Data Management Programming

Chengpu Wang
Independent Researcher
Chengpu@gmail.com

Abstract

Type expressiveness is defined in this paper as a convention to use public getters to reveal each type's information in a self-contained way so that its instance can be cloned, serialized and persisted by reflecting on the type definition only. Type expressiveness can be the foundation to allow data exchange between different address spaces created by the same or different programming languages.

A new behavior descriptive entity type called spec is proposed, which combines the traditional interface with test rules and test cases, to completely specify the desired behavior of each method, and to enforce the behavior-wise correctness of all compiled units.

The combination of type expressiveness and spec results in a new programming paradigm, which allows the separation programming space into 1) a behavior domain to aggregate all behavior programming in the format of specs, 2) a object domain to bind each concrete spec to its data representation in a particular address space, and 3) a realization domain to transfer data between the address spaces using the type expressiveness. Such separation guarantees the strictness of behavior satisfaction at compile time, while allows flexibility of dynamical binding of actual implementation at runtime. Such separation also allows a unique scheme for multiple inheritance which is resolved at compile time in the behavior domain.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Abstract data types

General Terms Computer Programming Language, Programming Paradigm, Serialization

Keywords Computer Programming Language, Programming Paradigm, Serialization

1. Introduction

The evolution of programming languages [1] can be viewed as a progress on how well an object can express itself. Assembly languages [3] such as Intel 8086 [4] series add symbolic instructions and method call to machine language. Procedural languages [2] such as C [5] add data type, API and method callback to assembly languages. Object-oriented languages [2] such as C++ [6] add dynamic cast and interface to procedural languages. Reflective languages [7] such as Java [8] add reflection and the ability to enumerate implemented interfaces to object-oriented languages. Dynamic programming languages [9] such as C# [10] add capability to define and instantiate new type at runtime to reflective languages.

From behavior perspective, an object is no more than its collection of interfaces [2]. The ability to enumerate implemented interfaces in reflective languages is crucial for run-time query of objects, which is crucial for behavior-centric programming methodology [2]. For example, recent development of semantic web [11] allows a piece of code to be writing from behavior perspective rather than from intended usage perspective, and its usefulness is discovered by querying for different intended usage. The ability of dynamic programming allows adaptation of an existing behavior for a new use at runtime, which liberates the codes from prefabricated behaviors at compile time. For example, using C#, it is very easy to create a dictionary for any special key type and value type at run-time, as demonstrated by the following code fragment: `typeof(Dictionary<,>).MakeGenericType(new Type[] {keyType, valueType})`.

In the present time, a data object of predefined type seems to be a weak link in this behavior-centric and adaptive digital world. Due to its safety, robustness, efficiency and execution accuracy [12], statically compiled [2] strong-typed [2] programming languages such as C, C++, Java and C# still dominate the industrial digital world. In most such languages, a data type has to be precisely defined at compile-time, and any change of it requires recompilation. Recompilation is justified if such change brings behavior changes of the program, but it seems not very desirable when a purely descriptive change brings extensive recompilation all over a large system. For example, suppose a data type needs to accommodate a new member which will not affect any behavior of all programs, except at entrance and exit points. Such change can be easily accommodated in a database or in a data descriptive language [13], but it requires recompilation of all programs which uses the data type. Also, it may not be a good idea to mix implementation-specific types with behavior-centric logics, especially when a casual earlier decision may cause expensive later regrets, such as the famous Y2K problem [14]. Another example is the choice of using 32-bit integer type many years ago to represent stock trading volume, which brought down many trading algorithms in financial world when the daily volume of a stock exceeds 2G during the recent financial meltdown; due to the interweave between behavior programming and implementation programming, a simple replacement of 32-bit integer with a numerical type of larger capacity to record increased trading volume may result in nonsense outputs or runtime exceptions from all relevant `printf` or alike statements. Also due to the interweave between behavior and implementation in conventional programming languages, porting pieces of behavior codes from one programming language to another programming language, or just from one operating system to another operating

system, is frequently required and time-consuming yet tedious and error-prone. It would be nice if a runtime can tolerate difference of type definition without risking behavior changes, and it would be better if a programming language can have its behavior-centric logics clean from any implementation, including type definitions. This paper will propose a new programming language paradigm which:

1. Separate programming space into behavior domain, object domain and realization domain.
2. Centralize all behavior codes in the behavior domain at compile time using one language, while allow specific data binding codes written in different languages in the object domain at runtime.
3. Define a new behavior descriptive entity type called spec and use it for programming in the behavior domain, to guarantee the desired behaviors independent of implementation.
4. Make object domain as flexible as possible to allow runtime type definitions and definition changes, including using external type descriptive languages such as XML.

The separation between the implicit type usage in behavior domain and the explicit type definition & instantiation in object domain requires each data object to express itself in a self-contained way so that it can be instantiated, cloned, transferred and persisted without the need for Human interpretation and intervention, in contrast to the need to implement certain interface for cloning and certain other interface for serialization in Java [8]. The first part of this paper will introduce the concept of type expressiveness in the form of a coding convention, and demonstrates its usefulness by using it to achieve improved cloning and serialization over the default .NET XML serialization [10].

2. Type Expressiveness

2.1 Definition

Type expressiveness is defined as using public getter and related attributes to reveal each type's information in a self-contained way, so that sufficient information to clone the instance can be obtained by querying public getters in a systematic manner. If a type is *fully expressive*, the clone will be identical to the source. Although type expressiveness is demonstrated using C# in this paper, same programming principles can be applied to other existing reflective languages such as Java and C++. This paper uses the terminology of C# exclusively, such as `attribute` for `annotate`, and referring to getter and setter in the format of property, e.g., without the "get" and "set" prefix.

A few types are predefined to be *primitively expressive*, whose values can be converted between the corresponding string values:

1. A C# string is primitively expressive.
2. Any C# primitive type [10] is primitively expressive. Each primitive type implements 1) an instance `ToString()` method to convert its value to a string, and 2) a static `Parse()` method to construct an instance of the type from a value-coded string.
3. Any C# value type [10] is assumed to be primitively expressive if the type 1) has override `ToString()`, and 2) has a static `Parse()` method that accepts a string parameter and return a instance of the value type. This category includes all the C# enum types. It also includes other commonly used C# value types such as `DateTime`. Because type expressiveness is only a convention, whether `Parse()` can turn all the strings returned by `ToString()` into correct values is not checked for each such value type.

The following are some common good practices regarding to data encapsulation in C# [15]

- Field should always be private or protected in a class unless the field is a public constant.
- Field should usually be accessed through getter and setter. Public setter should only exist if it is absolutely necessary to do so.
- Field should be read-only whenever possible. A read-only field is assigned either in constructor or when the field is defined and initiated, and its value is no longer allowed to change after the initial assignment.
- To better handle exception, sometimes the construction of an object is divided into two steps: 1) calling a constructor, and 2) calling one or several initiation methods immediate after calling the constructor.

Following the above common good practice, a type is expressive if its public getters are sufficient to reveal how it is constructed, namely:

1. It has either a public default constructor, or an expressive constructor. An *expressive constructor* is a public constructor with each parameter matching a public getter. The matching requires that 1) the type of the getter equals the type of the parameter, and 2) the names of the getter and the parameter matches according to predefined matching rules. The default matching rules require that the two names have to be identical after masking off camel case, type decorator, and collection decorator.
2. It may have some expressive initiation methods. An *expressive initiation method* is a public method which 1) receives no parameter, and 2) is tagged by the `[InitSeq(int)]` attribute. The parameter in the `[InitSeq(int)]` attribute defines the calling sequence if there are multiple initiation methods.
3. It may have some instance expressive setters or static expressive setters. An *expressive setter* is a public setter which 1) has a corresponding public getter, and 2) is not tagged by the `[NonExpressive]` attribute.
4. It may have some instance expressive getters or static expressive getters. An *expressive getter* is a public getter which 1) has no corresponding public setter, 2) is not tagged by a `[NonExpressive]` attribute, and 3) is a collection implementing the `ICollection` interface. An expressive getter is used to serialize the elements of the collection.

The above information of an expressive type is collected using C# reflection. However, reflection is not the only mechanism for type expressiveness. For example Java reflection mechanism does not provide parameter name so that Java system library introduces the widely used `ConstructorProperies` attribute [16] to manually bind the constructor parameters to the public getters. For languages that do not have reflection capability, the expressiveness of a type may be discovered during compilation process by inspecting the source code directly.

Figure 1 displays a simple example of a fully expressive type definition, which also shows that expressiveness of a type can be used to document the behavior of the type from the perspective of the minimal information to recreate an instance of the type, e.g., the `[NonExpressive]` attribute suggests that an array getter will not expose its underlying collection, and the two `[InitSeq]` attributes suggest the dependency relation between the two initiation methods.

```
interface MyIntface {
    int MyValue { get; set; }
    bool ConnectToDB();
}
public class MyExpressiveType : MyIntface {
     // an internal field that can construct itself needs not be expressive
     private static readonly ILog myLog = LogManager.GetLogger("My Log");

     // an expressive setter
     public double MyValue { get; set; }

     // an expressive getter
     public List<double> MyCollection { get { return myCollection; } }
     private readonly List<double> myCollection = new List<double>();

     // a non-expressive getter
     [NonExpressive]  // a marker for non-expressiveness
     public double[] MyArray { get {
        double[] clone = new double[MyCollection.Count];  myCollection.CopyTo(clone, 0);  return clone; } }

     // an expressive constructor
     public MyExpressiveType(double myReadonly) {  // match to getter not field
         this.myConnectionString = Encryptor.Decode(myReadonly); }
     public double MyReadonly { get {  // use camel case
         return Encryptor.Encode(myConnectionString); } }
     private readonly string myConnectionString;

     // two expressive initiation methods
     [Init(1)]  // return type suggests that first call can fail
     public bool ConnectToDB() {
         myConnection = DB.Connect(myConnectionString, MyValue);
         return myCollection != null; }

     [Init(2)]  // second call depends on previous successful calls
     public void SetDBLogger() { myConnection.SetLogger(myLog); }

     private DBConnection myConnection;
}
```

Figure 1. An Example Expressive Type

2.2 Default Naming Convention for Constructor Parameters

One of the major efforts to make a type expressive is to match the name of each parameter in an expressive public constructor with a name of a public getter. Such name matching already exists in common practices, such as the camel naming convention [15] between property and field, or the usual identical name between a constructor parameter and a read-only field [19]. These common practices are formalized as the default naming convention for the expressive constructor parameters, in which a parameter name is allowed to have the following optional difference with the corresponding getter name:

- Camel case: The first letter of a parameter name can be the small letter of the first capital letter of a getter name.
- Type decorator: An extra character can precede the name of a parameter to decorate the type of the parameter, such as `"i"` for signed integer, `"u"` for unsigned integer, `"d"` for double, `"w"` for string, `"b"` for Boolean.
- Array/Collection decorator: An extra `"s"` can follow the name of a parameter to indicate that the parameter is an array or a collection.

After decorators are removed, and camel case is corrected, the parameter name matches a getter name if they are case-sensitive equal.

2.3 Serialization of Expressive Types

A 8-bit char stream seems to be the only data format that can be ubiquitously understood among the different languages and/or among on different operating systems [2]. With a proper ending protocol, a string can be translated among the different languages and/or among different operating systems. Thus, instance of string can be regarded as the basic transferable elements in a stream between a sender and a receiver in general cases of serialization [17]. All strong typed languages tend to have similar set of primitive types, and the same set of string representation of these primitive types [2]. Because all other types are made from string and primitive types [2], instances of any type can be coded into the stream. In additional to instance data, an *expressive stream* also contains type information and name for each type instance in the stream. The expressive stream may contain the following types of stream items:

- `VString` packs an instance name with a string.

- `Prim` packs an instance name with a value of primitively expressive type other than a string.
- `Value` packs an instance name for a stack object.
- `Refer` packs an instance name with a hash code for a heap object. A zero hash code means that the instance is a null heap object.
- `TypeInfo` packs a type name.
- `IntfInfo` packs a type name with an interface name.

In the future, the expressive stream may also contains the `Code` type of stream items which embed codes for initiation methods in the same way as a HTML document embeds JavaScript code [19].

The order of stream items for an instance of non-primitively expressive type is the following in an expressive stream.

1. A `TypeInfo` item for the type.
2. A `IntfInfo` item for each implemented interface of the type, which exist at most once for each type.
3. A stream item for each static expressive setter, which exists at most once for each type.
4. Stream items for each collection item of each static expressive getter, which exist at most once for each type.
5. A `Value` item for each stack object or a `Refer` item for each heap object.
6. A stream item for each instance expressive setters.
7. A stream item for each expressive constructor parameter.
8. Stream items for each collection item of each instance expressive getter.

Such scheme applies regressively to instances of constructor parameters, expressive setters and collection items of expressive getters, until all objects are reduced to instances of primitively expressive types.

2.4 Type Binding in Expressive Stream

The above step 1 to 4 in the serialization process applies *type binding* to each instance of the type which is not primitively expressive type. To apply type binding to an instance of a primitively expressive type, a `TypeInfo` item precedes the instance. In the complete format, an expressive stream applies type binding to all type instances which it contains.

2.5 Name Binding in Expressive Stream

In *name binding*, the name for each stream item is derived from either type name or instance name using the following conventions:

- `<type name>:<interface name>` identifies the name of an interface for the type.
- `<type name>.<setter name>` identifies the name of a static expressive setter for the type.
- `<instance name>.<setter name>` identifies the name of an instance expressive setter for the type instance.
- `<instance name>$<getter name>` identifies the name of an instance getter which acts as a parameter in an expressive constructor for a type instance.
- `<type name or instance name>*<getter name>` identifies the name of a static or instance expressive getter for a type when the expressive getter is either an array or an `IList` collection.
- `<type name or instance name>@<getter name>` and `<type name or instance name>&<getter name>` identify the name of a static or instance expressive getter for a type when the expressive getter is a `IDictionary` collection and the stream item is the key and value for the `IDictionary` collection, respectively.

Name binding also applies regressive to each stream item.

2.6 Deserialization of Expressive Types

Type and name binding in expressive stream is necessary because the receiving end of the stream may be written by a different software version or even a different language. Using type and name binding, a complete data set of an instance of a non-primitive expressive type is retrieved and reordered from an expressive stream. The instance is then reconstructed in the following order:

1. Check to see if the system has the serialized type defined. If so, match the serialized type with the defined type; otherwise, the type needs to be created dynamically.
2. If the type's interface set has never been checked for the type, check its interface set to make sure that the type has identical interface set in the stream.
3. If the type's static expressive setters have never been called for the type, call the static expressive setters.
4. If the type's static expressive getters have never been populated for the type, populate the static expressive getters.
5. Call the type's expressive constructor.
6. Call the type's instance expressive setters.
7. Call the type's expressive initiation methods in sequence. If a method returns false, stop the construction, and throw an exception. Other return types of the initiation method are ignored.
8. Call the type's instance expressive setters again.
9. Populate the type's instance expressive getters.

Such scheme applies regressively in the order exactly opposite to the serialization process.

To better handle different versions of a same type definition, a `def` method may be introduced in addition to the `get` and `set` methods for property, which hold the default value of the property.

2.7 Reference Integrity

All objects are cloned after a serialization and de-serialization cycle. A heap object is access through a reference, which has one-to-one correspondence with hash code for the heap object in C# and Java. If there are multiple references to a heap object, the reference relationship is preserved among de-serialized objects. This is called the *reference integrity* [18].

To achieve reference integrity, all serialized references are saved in a table indexed by the hash codes. During serialization, if a reference is not in the table, a full heap object follows the `Stream.Refer` item in the stream; otherwise only `Stream.Refer` item exists in the stream. During de-serialization, when a `Stream.Refer` object is fetched from the stream, the table is populated if it does not contain the reference, and all subsequent `Stream.Refer` object in the stream gets the reference from the table.

2.8 Format Difference of Expressive Streams

An expressive stream is not always in the complete format. When the sender and the receiver shares the same type definition, the member types of each type is known on both ends of the stream so that the type binding may not be necessary for the members whose types are sealed. The order of the stream is also known on both ends of the stream so that the name binding for members may also be omitted. The simplified expressive stream is much more efficient

in transferring type instances between the sender and the receiver. A special case is that the sender is the receiver, which results in cloning.

The format of each expressive stream is set when the stream is constructed, so that each expressive stream has only one format.

2.9 Checking for Full Expressiveness

If after an instance of a type is cloned using expressive stream, and the clone instance equals the original instance, the type is *fully expressive*.

Instead of overriding `Equals()` for each type, the concept of type expressiveness can also be used to compare the two instances after cloning. Using reflection, each object can be decomposed into fields of primitively expressive type, and then compared field-by-field. The comparison is carried out recursively to all the fields and all the base classes unless specified otherwise:

- If a field is tagged with the `[CompareIgnore]` attribute, it is ignored for comparison.
- If a base class is tagged with the `[CompareBase]` attribute, the comparison will not go deeper than that class. A child class can also use the `[CompareBase(String)]` attribute to specify the base class to stop the comparison.

It turns out that C# system library contains some implementation-dependent behavior on the field level. As an alternative, the comparison can be performed on public getters only. Since type expressiveness is only a convention, it is programmer's responsibility to decide which of the three equal comparison methods to use.

2.10 A Persistence Example of Expressive Types

Data frequently needs to be in delimited format to be exchanged with other data processing programs, such as the Microsoft Excel and the R Statistics Package. Thus a delimited format for serialization is quite useful. An expressive and serializable tab-delimited text file contains the following three special types of lines in addition to normal text:

1. *type line*: "#?\t<Type1>\t<Type2>...\n". A type line must leads any other formatted text lines.
2. *name line*: "#@\t<Name1>\t<Name2>...\n". A name line must follow a type line, with the delimited fields in both lines matched by the column positions.
3. *data line*: "?\t<Datum1>\t<Datum2>...\n". One or multiple data lines must follow each pair of type and name lines. The type and name of each delimited field in the data line is provided by the corresponding delimited fields in the type line and the name line of the same column position.

Figure 2 shows the tab-delimited file format for an instance of the example expressive type of Figure 1

An expressive stream contain all information to generate the tab-delimited text files, except the format information, such as how to break stream into lines and files, and in what sequence to write the stream into files. Such decisions should be made by the Human user who generates the files. Because the de-serialization process reorders the fully expressive stream according to type and name binding, it can easily handle expressive stream in any persisted formats.

2.11 Summary

The major improvement of the serialization using type expressiveness over the default .NET serialization functionality is to pick an expressive constructor over the default constructor, while the default .NET serialization functionality 1) always constructs an object using the default constructor, and 2) ignores all the getters without the corresponding setters. Such improvement encourages the usage of read-only properties in C#. The code for type expressiveness serialization is published at http://sourceforge.net/projects/expressivetype/.

If mapping of basic types between two computer languages are provided, the expressive stream created by one computer langauge can be read by another language. For instance, a configuration file in a standard format may be used to equalize the `System.Int32` type in C# [10] to the `int` type in Java [8] and the `long` type in C++ [6], so that the string representation generated in one programming language can be translated to the correct value in another programming language. Thus, the expressive stream can be used as a mechanism for inter-language data communication.

2.12 Discussion

The example of the tab-delimited formatted stream shows that an end-usage case of data objects may contain enough information to generate an expressive stream. When a type definition in an expressive stream is not known, its definition can be generated dynamically. For example, the persisted expressive stream format in Figure 2 contains all information to generate the `MyExpressiveType` type in Figure 1. However, if an executable can only understand predefined types, such dynamically generated types are only of limited usage. An object is de-serialized according to its type; while an object can be used according to its interfaces [2]. For example, both Java and C# encourage use container interfaces instead of directly accessing the corresponding container implementations [8][10]. Thus dynamically generated types can be very useful if data instantiation can be separated from data usage. Such requirements call for a new programming paradigm [21] that clearly separate object management and object behavior. For simplicity of discussion, a new imaginative programming language named `Expresso` is introduced in this paper to describe this new programming paradigm.

3. A Proposed New Programming Paradigm

3.1 Interface, Trait and Spec

A software *interface* [22] is a what-to-call contract between a client and a service provider usually in the format of a set of method signatures. However, interface itself can only specify little desired behavior in code, while relying heavily on documentation for its functionality. For example, an interface for addition may only specify the possible result type and the possible operand types. The disparity between the desired behavior and the actual behavior is usually a major source of software bugs.

A *trait* [23] contains implementations in addition to the what-to-call contract of an interface. In this respect, an abstract class can also be viewed as a trait. The implementation in trait may contain more behavior specification than the function signature of addition, but it may still be insufficient to specify what addition is.

A *spec* is defined here as a trait with specified test cases and test rules. For example, a spec for addition may be built on the trait for addition, and further requires that:

- The result of adding a first operand to a second operand equals the result of adding the second operand to the first operand.
- The result of adding two positive operands is larger than any of the operands.
- If one operand is a positive integer, the result of the addition equals incrementing the other operand by the integer count.

An object implements a spec by 1) implementing all the what-to-call contracts, and 2) passing all test cases and test rules at compile time. Full coverage automatic unit-tests [24] during program building processing have been the highest quality standard for correctness and consistency of modern software engineering. A spec

```
#?  UnitTest.MyExpressiveType   UnitTest.MyIntface          System.Double                 System.Int32
#@  MyExpressiveType           MyExpressiveType:MyIntface  MyExpressiveType$MyReadonly   MyExpressiveType.MyValue
?   19342748                   UnitTest.MyExpressiveType   12345                         6789
#?  System.Double
#@  MyExpressiveType*MyCollection
?   0.123
?   456.7
?   890
```

Figure 2. The Tab-Delimited File Format for an instance of the Example Expressive Type of Figure 1.

can be viewed as a trait with integrated unit-test so that it has higher code quality in specifying the desired behaviors than either an interface or a trait.

3.2 Domains of Operations

The operations by any computer program can be divided into three domains:

- *Object domain*: This domain focuses on the initiation, destruction, cloning, passing, transmitting and persisting of individual objects, but not on how objects interact with each. As shown previously, type expressiveness can automate these operations.
- *Behavior domain*: This domain focuses on how objects interact with each, and what are the expected behaviors for the set of exposed method of each object. Interfaces and traits rather than type have been preferred method to specify behaviors of an object [2]. Spec should be a better specification. The question is if any behavior can be specified by implementation-neutral specs without using any type. When arithmetic operations are performed, the actual value of an arithmetic operand is never used directly but only indirectly through the comparison and the arithmetic relations. This seems to be a general rule for programs in behavior domain. So it may be quite possible to specify behavior using implementation-neutral specs only.
- *Realization domain*: This domain focuses on how a behavior specification in the behavior domain binds to a concrete object in the object domain. Such binding is hard-coded when a type which implements an interface is instantiated directly to provide the services of the interface in a traditional programming language. A preferred behavior realization is to use the abstract factory pattern [25], which provides an implementation to an interface as the interface only. `Expresso` uses specs instead interfaces, and uses the abstract factory pattern *exclusively* to provide the spec implementation. In this way the codes to manage objects are clearly separated from the rest of the codes, with the former in the object domain and the latter in the behavior domain. The realization domain connects the two very different domains of operations, and embeds each entity with dual roles. For example, a spec may appear in a method signature in the behavior domain, while a heap pointer of the underlying type may actually be copied into the method in the object domain.

The design philosophy of `Expresso` is to be as strict as possible in the behavior domain, while as flexible as possible in the object domain, relying on spec testing in the behavior domain to validate implementation in the object domain. In `Expresso`, codes in the behavior domain are strong typed and statically compiled, while codes in the object domain are dynamically linked. Such flexibility in the object domain allows runtime creations and changes of the underlying data type definitions, while such strictness in the behavior domain guarantees the correctness and consistency of the software behaviors.

3.3 Spec Examples

In grammar, a spec looks like a class in an object-oriented language, e.g., each spec instance has a `this` member to refer to itself, the spec may have public, protected and private access levels for its methods and fields, and a child spec can inherit from a parent spec. A spec can contain *abstract method* which is declared but not defined method. A `ISpec` spec serves as the base spec for all the other specs in `Expressio`. Similar to an Object class in C#, `ISpec` declares an abstract `IsEqual()` method, as well as two testing rules for the method, which are shown in Figure 3. Each `[rule]` attribute precedes a testing rule callback method: `Self()` binds to `this` and `Reciprocal()` binds to `IsEqual(ISpec)`. At the final stage of compilation, `Self()` is called for each instance of `ISpec` and `Reciprocal(ISpec)` is called whenever `IsEqual(ISpec)` is called. The compilation fails if any rule method returns false. In this paper, C# styled attributes are used for `Expresso` compiler directive.

`ISpec` spec has another abstract method `ToString()` to help identify an instance of ISpec, which is shown by Figure 4. `[rule(this, ISpec other)]` binds `Reciprocal()` test to this and a other instance of `ISpec`. Both `IsEqual()` and `ToString()` contain the `Reciprocal()` rules of the same signature, which is allowed because a test rule is not considered as part of `ISpec`, instead it belongs to the method under test, e.g., `[rule(this)]` belongs to `this`, which is a special method of `ISpec`.

Before a spec can generate test cases, it needs the ability to be instantiated. Unlike class constructors, the *spec constructors* are entry-points for the abstract factory [25] in the object domain. A spec constructor which accepts a string as its only parameter is named as a *literal constructor* in this paper, which allows the syntax `<Spec> <instance>=<literal>` in the code to instantiate the spec. The literal constructor of `ISpec` is declared in Figure 5. When `ISpec(String)` is called during realization test,the attribute `[case(this)]` generates a test case for all the `[rule(this)]` test rules. The attribute `[rule(ISpec(const String), ISpec(const String))]` binds `Mutual()` test to each mutual pair of multiple `ISpec(const String)` calls. `ISpec` also declares a destructor, a default constructor and a copy constructor with test cases and test rules respectively, as shown in Figure 6.

`Bool` implements Boolean arithmetic, and it is a spec derived from ISpec. `Bool` has two constants defined in Figure 7.

Because `Bool`'s constructors create instances of `Bool`, some of them need to be defined in the object domain. For the same reason, the destructor of `Bool` also needs to be defined in the object domain. The constructors and the destructor of `Bool` are shown in Figure 8, in which the `[object]` attribute states that the definition of `Bool(String)` exists in the object domain, and the `[override(ISpec(String))]` attribute states that `Bool(String)` takes up all the test cases and test rules of `ISpec(String)`.

The implementation of `IsEqual()` also needs to be defined in the object domain because how to compare two `Bool` instances is implementation-dependent. For example, each Bool instance can

```
[abstract] public Bool IsEqual(const ISpec other) const;
[rule(this)] Bool Self() { return this.IsEqual(this); }
[rule(IsEqual(const ISpec))] Bool Reciprocal(const ISpec other) {
    if(this.IsEqual(other)) return other.IsEqual(this));
    if(other.IsEqual(this)) return false;
    return true; }
```

Figure 3. The Declaration for ISpec.IsEqual().

```
[abstract] public String ToString() const;
[rule(this, other)] Bool Reciprocal(ISpec other) {
    if(this.IsEqual(other)) return this.ToString().IsEqual(other.ToString());
    return true; }
```

Figure 4. The Declaration for ISpec.ToString().

```
[case(this)] [abstract] public ISpec(const String literal);
[rule(ISpec(const String), ISpec(const String))] Bool Mutual(const String a, const String b) {
    if(a.IsEqual(b)) return ISpec(a).IsEqual(ISpec(b));
    return true;  }
```

Figure 5. The Declaration for ISpec.ISpec(String).

```
[case(this)] [abstract] public ISpec();
[case(this)] [abstract] public ISpec(const ISpec other);
[rule(ISpec(const ISpec))] Bool Self(const ISpec other) { return ISpec(other).IsEqual(other); }
[rule(ISpec(const ISpec), ISpec(const ISpec))] Bool Mutual(const ISpec a, const ISpec b) {
    if(a.IsEqual(b)) return ISpec(a).IsEqual(ISpec(b));
    if(ISpec(a).IsEqual(ISpec(b))) return false;  // a != b
    return true; }
[abstract] public ~ISpec();
```

Figure 6. The Declaration for ISpec.ISpec(), ISpec.ISpec(ISpec) and ISpec.~ISpec().

```
[case(this)] public static const Bool true("true");
[case(this)] public static const Bool false("false");
[rule(this)] Bool Exclusive() {
    if((this.IsEqual(true)) return true;
    if((this.IsEqual(false)) return true;
    return false;  }
```

Figure 7. The Definitions for Bool Instances.

```
[override(ISpec(const String))] [object] public Bool(const String literal);
[override(ISpec(const ISpec))]  [object] public Bool(const Bool);
[override(ISpec())] public Bool() : Bool(false) {}
[override(~ISpec())] [object] public ~Bool();
```

Figure 8. The Definition of Bool.Bool(String), Bool.Bool(Bool) and Bool.~Bool().

```
[override] public String ToString() const {
    if (this.IsEqual(true)) return "true";
    return "false"; }
[case] Bool () { return true.ToString().IsEqual("true"); }
[case] Bool () { return false.ToString().IsEqual("false"); }
```

Figure 9. The Definition of Bool.ToString().

be implemented in the object domain as either an integer (as in C/C++) or a bit.

Although `Bool` is derived from `ISpec`, `ISpec.IsEqual(ISpec)` returns `Bool` already, because the *method signature* in `Expresso` is defined as the method name and the list of the specs of its parameters, so that the return type of the method is excluded from the method signature. Such seemly peculiarity also exists in other languages, such as between `Bool` and `Object` types in C# [10]. The real problem here is the existence of the `if` flow control before `Bool` is defined. To let the behavior domain has conditional flow controls before `Bool` is defined, `Expresso`'s behavior domain relies on its object domain for the definition of the following two basic flow controls, 1) the `while` flow control, and 2) the `break` flow control, which can be used to construct all other flow controls [2], such as the `if` flow control.

In addition to enforce correctness, test rules also help implementation of specs. For example, the `Exlcusive()` rule limits `Bool` instances to be either `true` or `false` but nothing else, which makes the implementation of the abstract methods of `ToString()` very simple, as shown in Figure 9. The `[case]` attribute precedes a test case, which is implemented by an anonymous method.

Having only two constructors, one destructor and one method defined in the object domain, `Bool` represents a behavior-intensive spec. In contrast, as the other spec referenced by `ISpec`, `String` represents an implementation-intensive spec, because different languages even different implementations of the same languages can have very different implementation model for their `string` type definitions [6]. It is possible that `Expresso` supports a variety of string implementations simultaneously, each of which binds to the `String` spec by a different parameter set in the factory, such as selecting a string implementation optimized for short size with constant content, or a string implementation optimized for long size with variable content. So the `String` spec may contain mostly test rules and test cases, while leaving most implementation to the object domain. Another difference is that unlike `Bool`, exclusive test cases can not be generated for each method of `String`, so that typical test cases may have to be generated according to how each method of `String` is implemented in the object domain, e.g., if string concatenation is implemented different between two long strings vs. between two short strings, both cases needs to be included.

3.4 Spec Hierarchy

The conventional class hierarchy [20] has little constraints on behavior specification. For an example, when an Apple class is to be derived from a Fruit class, there is little mechanism during compilation to guarantee that the Apple class behaves like an apple but not an orange. Such lax in definition may bring about unexpected behaviors at runtime. Another problem is that there is no compatibility test when combing parent classes to create a child class. For an example, methods of the same signature but different functionalities may exist in multiple parent classes, which result in the implementation-dependent inheritance behavior [6] or the diamond inheritance problem [26] of the conventional class hierarchy.

The spec hierarchy in the behavior domain is much stricter than the class hierarchy:

- A spec may contain either instance or static methods whose access level can be either public, protected or private to the spec.
- A spec uses test cases and test rules to specify the desired behaviors for its methods.
- The *specific name* for each method is <spec name>.<method name>, which uniquely identify the method. If a method has to be referred by its specific name in a context, it is a *specific method* in that context. A virtual method, including being an abstract method, can never be referred by its specific name.
- A child spec can be derived from one parent spec or multiple parent specs. The child spec inherits all the public and protected methods from all the parent specs, including all their test cases. If two specs conflict with each other in their tests, they can not be the parent specs for the same child spec; otherwise their child spec can never pass tests during compilation.
- A child spec can inherit multiple virtual methods of the same method signature from different parent specs. If all the inherited virtual methods have the same specific name, they are considered as one virtual method in the child spec, with the tests from all the parent specs combined for such a method in the child spec; otherwise, the child spec inherits an abstract method of the same signature, which the child spec may override. Thus diamond inheritance problem [26] is avoided. From business perspective, a method is like a clause in a contract set [28] so that each method shall have only one definition everywhere at least for each spec.
- A child spec can redefine an inherited virtual method. The new definition has to pass all the existing tests in the parent specs as well as all the new tests which the child spec may add to the method. Using the `[override]` attribute, the method signature of the original method can be replaced by the method signature of the new definition whose argument specs are either the same or the child spec of the the argument spec of the original method signature. For example, to avoiding slicing, in Figure 8, `ISpec(const ISpec)` is redefined as `Bool(const Bool)` for `Bool`, so that calling `ISpec(const ISpec)` on a `Bool` instance actually clone the `Bool` instance. In `Expresso` the actual parameter type determines which copy constructor is called, because in the object domain the copy constructor is usually treated as a clone request using type expressiveness. The override attribute in Figure 8 states that the copy constructor for Bool follows this convention.
- A child spec can restrict the access level for an inherited method, such as from protected to private. When a method is inherited from multiple parent specs, the access level for the method is the most restrictive among the parent specs.
- A child spec can add new tests for a method defined in any of its parent specs. The parent specs do not need to pass the new tests defined in the child spec.
- The `[violate]` attribute allows a spec to ignore either a test rule or all the test rules of a method. If the `[violate]` attribute is applied to a spec, all tests of the spec will be ignored. Preferably the `[violate]` attribute should only be used for the debugging purposes.
- A spec may contain either instance or static private fields. A parent spec uses getters and setters to expose its fields to its child specs.

A *method table* maps each method signature to a method definition, or nothing when the method is abstract. The method table also contains other characteristics for each method, such as the specific name and the access level. A specific method has the spec name as part of the method signature in the method table.

- Each spec has a corresponding method table called a *native table* which maps all its methods. The native table is built following the inheritance generations, so that the definition provided by a child spec always overriding the definition provided by its parent spec.

Signature	Definition
`IsEqual(const ISpec)`	`Bool Bool.IsEqual(const ISpec)`
`ToString() const`	`String Bool.ToString() const`
`ISpec(const String)`	`Bool Bool.Bool(const String)`
`ISpec(const ISpec)`	`Bool Bool.Bool(const Bool)`
`ISpec()`	`Bool Bool.Bool()`
`~ISpec()`	`Void Bool.~Bool()`

Figure 10. The cast table for `(ISpec)Bool`.

- When a child spec is cast to a parent spec, a method table called a *cast table* is constructed by mapping each method signature in the native spec table of the parent spec to the corresponding method definition in the native spec table of the child spec. For example, Figure 10 contains the cast table when `Bool` is cast to `ISpec`.

Each spec is characterized by a *parent table* which maps each of its parent name to a corresponding cast table. The native table of the spec always occupies the first row in the parent table. When a spec is cast to anther spec, the corresponding cast table is fetched from the parent table.

3.5 Resolution of Method Overriding

Method overriding is traditionally implemented using v-tables, which looks for the method definition by the virtual method signature for each class [29]. A cast table can function as a corresponding v-table. In contrast, `Expresso` implements method overriding using a new technique called c-table, which looks for the method definition by class for each virtual method signature. To demonstrate this, Figure 11 shows the C# codes which rely on v-table, while Figure 12 shows the C# codes which relies on c-table. The following are steps to implementing Figure 12 starting from Figure 11, assuming method overriding has been resolved using cast tables on spec level:

1. Create a delegate for each virtual method in the parent spec, such as `B._DF(B)`. By passing `this` reference as an additional first parameter, the delegate is a static version of the original virtual method.
2. Create a *c-table* for each virtual method in the parent spec, which maps the spec types to instances of the delegate, such as `IDictionary<Type, _DF> B._mapF`.
3. Change each definition of the virtual method to a specific method, so that the `virtual` keyword before `B.F()` and the `override` keyword before `D.F()` are deleted, and both methods are labelled by the `[specific]` attribute. Also, the delegate-compatible methods `B._F(B)` and `D._F(B)` are created to shadow `B.F()` and `D.F()`, respectively.
4. Insert the actual definition for the virtual method into the c-table in the static constructor of the parent spec and each child spec.
5. Provide a definition to the virtual method name, which looks up the c-table for the actual method definition. In Figure 12, the `[virtual(F())]` attribute maps `B._VF()` to the virtual method `F()` of `B`.

Similar techniques to optimize v-table from a lookup table into a index table [29] can also be used to optimize the c-table. Thus, the runtime speed of resolving method overriding using either technique is comparable. However, the c-table offers the following advantages:

1. Because each c-table collects all the overriding definitions of a virtual method together, it may allows better compile-time optimization. For example, the c-table lookup code inside the virtual method definition can be replaced with a branch control, with all the overriding definitions possibly inlined in the method.
2. Because each instance no longer carries a v-table pointer, it separates data operation clearly from data definition, which is in line with the programming paradigm of `Expresso`. If every instance method is transformed to a corresponding static method which passes the structure containing the instance fields as the first parameter, `Expresso` can be implemented by a procedure language which is not even object oriented, e.g. C.
3. The method resolution using c-table also allows static method to be overrided, which may be used to move reflection methods to be implemented all at compile-time.

```
class B {
  public virtual String F() { return "B.F()"; }
}
class C : B {
}
class D : B {
  public override String F() { return "D.F()"; }
}
class E : D {
}
```

Figure 11. Typical C# inheritance codes which use v-table at runtime to implement method overriding.

```
class B {
  [specific] public String F() { return "B.F()"; }
  protected static String _F(B b) { return b.F(); }
  protected delegate String _DF(B b);
  protected static readonly IDictionary<Type, _DF>
    _mapF = new Dictionary<Type, _DF>();
  [virtual(F())] public String _VF() {
    return _mapF[GetType()](this); }
  static B() { _mapF.Add(typeof(B), new _DF(B._F)); }
}
class C : B {
  static C() { _mapF.Add(typeof(C), new _DF(B._F)); }
}
class D : B {
  [specific] public String F() { return "D.F()"; }
  protected static String _F(B b) { return ((D)b).F(); }
  static D() { _mapF.Add(typeof(D), new _DF(D._F)); }
}
class E : D {
  static E() { _mapF.Add(typeof(E), new _DF(D._F)); }
}
```

Figure 12. The corresponding C# inheritance codes which use source code preprocessing at compile-time to implement method overriding. Each inserted identifier is lead with a _ and the identifier is not accessible outside.

3.6 Compilation and Linking Process

A *concrete spec* is a spec which is preceded with the `[concrete]` attribute. Only concrete specs can be instantiated in the behavior domain. `Expresso` compilation does not generate executable directly but implores other compilers called the *target compiler* to do so. The `[concrete]` attribute also specifies the targeted compiler of the compilation process of `Expresso`. For example, the `Bool` spec can be defined as:

`[concrete(C#)] spec Bool {...}`

`Expresso` is statically typed and statically compiled in the behavior domain, with most workload done in the compilation process, so that its behavior is very predictable at runtime. The `Expresso` compilation process is the following:

1. Generates a native table for each spec in the inheritance order. Meanwhile, populate the c-table for each corresponding virtual method.
2. Generates a parent table for each concrete spec. Meanwhile, generates a traditional interface called a *contractor* for each concrete spec, which is composed of all the methods which are preceded with the `[object]` attributes. For example, the contractor for Bool contains `Bool(const String)`, `Bool(const Bool)`, `Bool()`, `~Bool()` and `IsEqual(const ISpec)`.
3. For each spec instance, generate a *cast pointer* pointing to either the native table of the spec as default, or one of the cast tables of the spec.
4. Resolve casting by varying the cast pointer for each spec instance. The compilation process fails if a spec instance is cast outside the parent table of the spec.
5. Apply the specific name for each member reference. The compilation process fails if a member is referenced but not defined, or if the access control is violated.
6. Generates a *test table* for each concrete spec, which maps each method signature to a set of test rules and a set of test cases. The test rules and the test cases from the parent specs are all added to the corresponding sets in the concrete spec.
7. Adopts the spec codes to the targeted compiler and compiles the codes for the concrete specs in the behavior domain using the targeted compiler. For each concrete spec, the compiled result is called the *executor*, while the compiled result for the test table is called the *tester*.
8. Compiles the codes in the object domain against the contractor. The compiled result which binds to each contractor is called the *implementer* for the corresponding concrete spec.
9. Runs through all the test cases in the tester, which trigger all the test rules in a chain reaction. If any test fails, the compilation process also fails.

If a concrete spec has no contractor, the spec can be instantiate directly in the behavior domain; otherwise, the concrete spec is instantiated dynamically at runtime by binding to an implementer through its contractor. A contractor can be implemented by multiple implementers in the object domain, one of which can be chosen at runtime through either a parametrized spec constructor or a configuration file in the object domain. For an example, an a parametrized `String` constructor may choose among different string models for the intended usage. Such dynamic binding scheme also makes the debugging and enhancement of code easier. For example, a change of implementation such as addition of new members to a data object is transparent to the behavior domain codes.

Thus, `Expresso` is a general-purposed, statically compiled [2] strong-typed [2], dynamically linked [2], self-tested [24], multi-platform, multi-target and source-to-source [30] programming language.

3.7 The Requirements for the Targeted Programming Language

Different programming languages have different strengths, capabilities, libraries, histories, user groups and weaknesses [31]; so that it is hard to say which programming language is the best overall. Being a "multi-target" language, `Expresso` tries to gather the fruits from most existing programming languages. As a summary, a programming language needs to satisfy the following minimal requirements to be a targeted language for `Expresso`:

- Has its primitive types mapped to the primitive types of `Expresso` in the data domain, which is equivalent to the Java primitive types plus a few string types.
- Accept separate compilation units.
- Can implement interface.
- Has `while` and `break` flow control constructs.
- Can define new types;
- Can be dynamically compiled and linked.

In addition to the minimal requirements, a targeted programming language should have more capabilities to better serve its role in `Expresso`:

- Has other flow controls. In addition to the `while` and the `break` flow controls, `Expresso` uses other flow controls in the behavior domain, such as `continue`, `if...else if...else`, `switch`, `for`, `foreach`, `do...while`, and etc. Although all the above flow controls can be constructed from the `while` and `break` flow controls, `Expresso` can use the native version of a flow control if such control is already provided by the target language.
- Has reflection mechanism. Unlike C# and Java, C/C++ has insufficient reflection mechanism so that the information for type expressiveness in the object domain has to be extracted from the codes directly. Thus the existing C++ compilers have to be modified to generate expressive type information during compilation before they can be used as the target compilers for `Expresso`.
- Can generate an instance of any type at runtime: C# has dynamic programming [9] capability. In contrast, it may require to generate & compile C/C++ source code on the fly, and dynamically link to the newly compiled executable to instantiate a type at runtime.

`Expresso` uses a standard set of xml files to inquiry the additional capability of each target programming language, before it source-to-source compiles [30] any codes in the behavior domain.

3.8 Discussion

The main idea of the new programming paradigm seems simple and straight-forward: 1) to use the expressive stream to transfer objects among different address spaces; 2) to use spec instead of interface for behavior specification; and 3) to separate programming space into behavior domain, object domain and realization domain. More details needs to be discussed and laid out.

`Expresso` compiles its behavior domain source code into an executable which is determined by a target compiler, to combine the behavior domain codes with the object domain codes. There are a few reasons for this choice:

- All procedural high-level programming languages have more-or-less the same logic flow constructs [2], such as `if`, `while`, `for`, `switch`, and iteration over items of a container using either the `foreach` syntax or the iterator pattern [32]. `Expresso` abstracts all these flow controls in its behavior domain programming.
- All object-oriented high-level programming languages have similar class hierarchy [2], with very similar access control

[2]. The major difference lies in how multi-inheritance is implemented, weather it is allowed directly (as in C++ [6]), or indirectly using interface and delegate (as in C# [10]), or indirectly using interface and inner class (as in Java [8]). `Expresso` provides a unified multi-inheritance scheme at compile time without the diamond inheritance problem [26] in its behavior domain programming.

- Different languages have different memory models [33] and thread models [2]; while each memory model or each thread model has its pro and cons. To complicate the matter, each operating system [34] has its own thread model and preferred memory models. `Expresso` delegates such implementation details to the object domain, so that its behavior domain programming can remain implementation-neutral.
- Different languages have different strength and weakness [31]. Each commonly used language has a large pool of applications, and it is not realistic to try to replace such an existing programming language with something completely new. One good example is one of the earliest programming language COBOL [35], which still have many applications running in many critical places, even their source codes may have been lost or no longer understandable. By converting its behavior domain codes into one of the common language, `Expresso` hopes to capture more programmers by providing a smoother migration path to a common language that is always compatible with existing executables. For example, at very beginning, a spec may be no more than an interface plus some test cases for a class in the object domain which still contains a lot of behavior-centric codes; gradually, these behavior-centric codes are migrated from object domain to behavior domain in an effort to provide unit test for these codes. Because the compiled Expresso behavior code can be in the target language, such migration differs not too much from a normal refactoring [36].
- This model does not exclude defining `Expresso` language in the object domain in the future. When a programming language for a new programming paradigm is defined before it is used extensively, it tends to have design mistakes and defects that will prevent its wide acceptance. `Expresso` language can accumulate usage experiences when it acts as a cross compiler.

`Expresso` is just a concept at this moment. It still has a lot of important questions to be answered. For instance:

- It is not clear how to specify a multi-thread performance requirement using test rules and test cases of the specs. Threading in different programming languages on different operating systems is an implementation-intensive issue, e.g., it has been reported that Java threading is less efficient than C# threading on Windows platform due to JVM abstraction of threading on all platform, including those which actually do not support threading [37]. How to effectively abstract threading in behavior domain remains a challenge.
- It is not clear how to constraints the outcomes using the specs because test rules and test cases of the specs are essentially tools to specify what are the desired behaviors; however they are not good at rejecting the undesired behaviors or side-effects such as all the possible parasite behaviors under the influence of computer viruses [38].
- The static compilation and dynamic linking aspects of `Expresso` calls for a security scheme to reliably certify an implementer once the implementer has been tested against the contractor and the tester. Because the testing can be execution-intensive and time-consuming, runtime testing of the implementer may not be practical. Also, the test of an implementer can be done on a powerful development machine, while the target machine on which the implementer is deployed can be a machine with very limited resources, such as an embedded system. How to prevent an impersonator to mimic a legitimate implementer remains a challenge.
- In contract with the conventional unit tests [24], in which all test cases are activated sequentially, in `Expresso`, the test rules and test cases are arranged by methods so that each test rule may be activated multiple times for the same test case. How to activate the test rules more efficiently at compile time remains yet another challenge.
- It may be necessary to distinguish between compile time test and runtime test. with the former for behavior test on a build machine and the latter for performance test on a target machine.
- The separation of object domain and behavior domain may provide a new dimension of data abstraction. An expressive type can be generated dynamically from an expressive stream, which can exists as an actual stream, as a file of various formats, the content of a web page, or even as data in a database. The instance of the expressive type is manipulated according to different interfaces of the type in the behavior domain, each of which provide one slice of the expressive type. Thus, there may be no need to write code manually to represent each expressive type but to generate the code dynamically from 1) the data existence in the object domain and 2) the data usage in the behavior domain. For example, in java spring framework [39], each property in the bean file corresponds to a setter. From the perspective of type expressiveness, if the type of the property is also provided, the code for the setter becomes redundant, which can be generated dynamically at runtime. For example, the code in Figure 1 can almost be generated by the data in Figure 2. In the programming paradigm of `Expresso`, an object in the object domain represents a pure data object, which should be contains no or minimal behavior codes, or even be determined dynamically by the data in various expressive formats. Each data file can be validated by various testers before it is put into production so that this dynamic linking approach may be quite safe.

4. The Expressive Type Project

An open source project to demonstrate type expressiveness is published at http://sourceforge.net/projects/expressivetype/. It contains two projects:

- An `Expressive` project contains the core APIs for the type expressiveness.
- A `UnitTest` project contains the unit test of the core APIs. This project also serves as a usage example for the `Expressive` project.

4.1 SerialInfo Class

A `SerialInfo` object is associated with each type if the type has either a default constructor or an expressive constructor, or the type itself is a primitively expressive type. It is used to collect expressive information of a type in the reflection space. It is constructed in the following steps:

1. See if the type is either `String` or `enum`. If so, finish constructing the `SerialInfo` object.
2. See if the type is a primitive value type. If so, populate the public getter `"MethodInfo Parse"` and finish constructing the `SerialInfo` object.

3. Scan for the implemented interfaces of the type, as its public getter `"Type[] Interface"`;
4. Scan for its static expressive setters, as its public getter `"IDictionary<String, PropertyInfo> StatSetter;"`
5. Scan for its static expressive getters, as its public getter `"IDictionary<String, PropertyInfo> StatGetter"`;
6. Scan for its instance expressive setters, as its public getter `"IDictionary<String, PropertyInfo> InstSetter"`;
7. Scan for its instance expressive getters, as its public getter `"IDictionary<String, PropertyInfo> InstGetter"`;
8. Scan for an expressive constructor, or a public default constructor, and populate its public getters `"ConstructorInfo Constructor"` and `"PropertyInfo[] Param"` with the constructor information;
9. Scan for its initiation methods, as its public getter `"MethodInfo[] Init"`.

`SerialInfo` class has a method to return a `SerialInfo` object for each type:

```
static SerialInfo Get(Type);
```

It contains a public method to clone an object from its expressiveness:

```
Object Clone(Object);
```

It also contains a method that converts a type's full-name into a `Type` object:

```
static Type GetType(String);
```

4.2 Fieldwise and Propertywise Classes

Static class `Fieldwise` provides regressive field-wise comparison between any two objects, and prints out first difference between the two objects:

```
static String Diff(Object left, Object right);
static bool IsEqual(Object left, Object right) {
    return Diff(left, right) == null; }
```

Static class `Propertywise` provides such comparison on public getters of the two objects only.

4.3 Stream Class

`Stream` is an abstract class that serves as the base class for the definition of the expressive stream item types, which are defined in Section 2.3.

4.4 IStream Interface

The function of an expressive stream is represented by an `IStream` interface, which has the following queue method methods:

```
ValueType Next();
void Append(ValueType value);
```

Due to type binding and naming binding requirements during de-serialization, it provides the following look-ahead functions:

```
bool HasNext { get; }
String NextName { get; }
Stream NextValue { get; }
```

When constructing an `IStream` object, there is a choice if to serialize interfaces, initiation methods and static properties of each type:

```
bool IncludeStatic { get; }
```

There is a choice if to serialize sealed type for each object:

```
bool IncludeType { get; }
```

There is another choice if to use name binding:

```
bool IncludeName { get; }
```

4.5 Serial Class

Static class `Serial` has a method to serialize a object with a name to a stream:

```
static void Serialize(IStream stream,
    Object obj, String name);
```

It has another method to serialize a static class or a static portion of a class.

```
static void Serialize(IStream stream,
    SerialInfo info)
```

It has three public methods to de-serialize an object with a name from a stream:

```
static Object DeSerialize(IStream stream,
    out SerialInfo info, out String name);
static Object DeSerialize(IStream stream,
    SerialInfo info, out String name);
static Object DeSerialize(IStream stream,
    SerialInfo info, String name);
```

In addition, it has a method to de-serialize a static class or a static portion of a class:

```
static void DeSerialize(IStream stream,
    SerialInfo info)
```

By caching all serialized and de-serialized heap objects, `Serial` also maintains reference integrity among the objects in each stream. It has one public method to release the cache, which should be called once serialization or de-serialization is done:

```
static void Clear();
```

4.6 BufStream Class

`BufStream` class is an implementation of `IStream` interface with name binding in memory. Its constructor is the following:

```
BufStream(bool bIncludeStatic, bool bIncludeType);
```

4.7 StrStream Class

Static `StrStream` class converts a `IStream` object into or from a `String[]` array, so that the stream can be transmitted between two applications.

```
static String[] Seralize(IStream stream);
static IStream DeSeralize(String[] array);
```

The first item in the `String[]` array identify the property of the stream, such as which programming language generates the array, so that the full type names can be mapped to the equivalent full type names of another programming language.

4.8 TabStream Class

Class `TabStream` represents a tab-delimited stream, with one example provided in Figure 2. Its delimiter, type marker, name marker, data marker and an new line marker are all read-only properties:

```
TabStream(String delimater, String typeMarker,
    String nameMarker, String dataMarker,
    String newLine);
TabStream() :
    this("\t", "#?\t", "#@\t", "?\t", "\n") {}
```

Other formatting properties are dynamically changeable:

```
int MaxItemPerLine { get; set; }
bool StartNewLineForNonPrimitive { get; set; }
bool StartNewLineForCollection { get; set; }
```

`TabStream` provides a serialization method for object, which specifies lines of getters to output first as a `PropertyInfo[][]` array:

```
bool Serialize(TextWriter writer,
   Object obj, String name, PropertyInfo[][] lines);
```

It also has a serialization method for static class or a static portion of a class:

```
bool Serialize(TextWriter writer,
    SerialInfo info);
```

`TabStream` has a method to convert text containing serialized objects into an `IStream` object, so that the text can be de-serialized.

```
IStream DeSerialize(TextReader reader);
```

4.9 A Slim Unit Test Architecture

To be free from any particular unit test framework, the static class `UnitTest` in the project `UnitTest` provides basic ability to unit test using log. The `UnitTest` project depends on all the other projects. By convention, it test each class using a test class of the same name with a public static method `Test()`. Within each `Test()` method, the unit tests are carried out in the following steps:

1. Call `Category()` to setup a category of a series of tests. The stack frame is used to trace where a new category of test starts.
2. Call `Case()` to perform one test. The stack frame is logged if `Case()` fails.
3. Call `Comment()` any time to add comment to the log.

When all unit tests are done, call `Summary()` to tally the overall results, which has been broken down for each category.

It is the responsibility of the user to:

- Divide test cases into related categories. The testing statistics is provided for each category, and category also help to find where the test fails quickly.
- Call `Test()` of each testing class.
- Decide if to go on with the rest of the tests when a test case fails.
- Handle exceptions.
- Find where the test fails from the log and the context hint.
- Compare logs for compatibility test.

Thus, this unit test package is less than a diagnostic tool about where the code is wrong. Rather, it focuses on what is wrong.

Acknowledgments

References

[1] Steven R. Fischer, *A history of programming language*, Reaktion Books, 2003.

[2] Michael L. Scott, *Programming Language Pragmatics*, Morgan Kaufmann Publishers, 2006

[3] *Assembly language*, Wikipedia, http://en.wikipedia.org/wiki/Assembly_ languages

[4] *x86 assembly language*, Wikipedia, http://en.wikipedia.org/wiki/X86_ assembly_language

[5] Brian W. Kernighan, Dennis Ritchie, *C Programming Language*, Prentice Hall, 1988

[6] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 2000

[7] *Reflection*, Wikipedia, http://en.wikipedia.org/wiki/Reflection_ (computer_programming)

[8] Cay S. Horstmann, Gray Cornell, *Core Java*, Sun Microsystem Press, 1999

[9] *Dynamic programming*, Wikipedia, http://en.wikipedia.org/wiki/Dynamic_ programming

[10] Andrew W. Troelsen, *Pro C# 2005 and the .NET 2.0 Platform*, Springer-Verlag, 2005

[11] Haralambos Marmanis, Dmitry Babenko, *Algorithms of the Intelligent Web*, Manning, 2009

[12] *Prototype-based programming*, Wikipedia, http://en.wikipedia.org/wiki/Prototype-based_ programming

[13] *Data descriptive language*, Wikipedia, http://en.wikipedia.org/wiki/Data_ Definition_Language

[14] *Year 2000 problem*, Wikipedia, http://en.wikipedia.org/wiki/Y2k_ problem

[15] Bill Wagner, *Effective C#: 50 Specific Ways to Improve Your C#*, Pearson Educations, 2005

[16] *constructor properties*, Java API, http://download.oracle.com/javase/6/docs/api/java/beans/ConstructorProperties.html

[17] *Serialization*, Wikipedia, http://en.wikipedia.org/wiki/Serialization

[18] *Referential integrity*, Wikipedia, http://en.wikipedia.org/wiki/Referential_ Integrity

[19] *JavaScript*, Wikipedia, http://en.wikipedia.org/wiki/JavaScript

[20] *Runtime System*, Wikipedia, http://en.wikipedia.org/wiki/Run-time_ system

[21] *Programming paradigms*, Wikipedia, http://en.wikipedia.org/wiki/Programming_ paradigm

[22] *Protocol (object-oriented programming)*, Wikipedia, http://en.wikipedia.org/wiki/Interface_ (object-oriented_programming)

[23] *Trait (computer programming)*, Wikipedia, http://en.wikipedia.org/wiki/Trait_ (computer_programming)

[24] *Unit Testing*, Wikipedia, http://en.wikipedia.org/wiki/Unit_ testing

[25] *Factory method pattern*, Wikipedia, http://en.wikipedia.org/wiki/Factory_ method_pattern

[26] *Diamond Inheritance Problem*, Wikipedia, http://en.wikipedia.org/wiki/Diamond_ problem

[27] *Virtual inheritance*, Wikipedia, http://en.wikipedia.org/wiki/Virtual_ inheritance

[28] Robert W. Emerson, John W. Hardwicke, *Business Law*, Barron's Educational Series, Inc., 1997

[29] *Virtual method table*, Wikipedia, http://en.wikipedia.org/wiki/Virtual_ method_table

[30] *Source-to-source compiler*, Wikipedia, http://en.wikipedia.org/wiki/Source-to-source_ compiler

[31] *Comparison of programming languages*, Wikipedia, http://en.wikipedia.org/wiki/Comparison_ of_programming_languages

[32] *Iterator pattern*, Wikipedia, http://en.wikipedia.org/wiki/Iterator_ pattern

[33] *Memory model (computing)*, Wikipedia, http://en.wikipedia.org/wiki/Memory_ model_(computing)

[34] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall, 2005

[35] *COBOL*, Wikipedia, http://en.wikipedia.org/wiki/COBOL

[36] Code refactoring, Wikipedia, http://en.wikipedia.org/wiki/Refactoring

[37] M.H. Lutz; P.A. Laplante, *C# and the .NET Framework: Ready for Real Time?* IEEE Software, 2003 Vol 20-1, P74-80

[38] *Computer virus*, Wikipedia, http://en.wikipedia.org/wiki/Computer_ virus

[39] *Spring Framework*, http://http://www.springsource.org/